Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity

Ettore Merlo[1]

École Polytechnique de Montréal, Départment de Génie Informatique (DGI),
C.P. 6079, Succ. Centre Ville, Montréal, Québec H3C 3A7, Canada
ettore.merlo@polymtl.ca

Abstract. An original method of spectral similarity analysis for plagiarism detection in university project is presented. The approach is based on a clone detection tool called CLAN that performs metrics based similarity analysis of source code fragments. Definitions and algorithms for spectral similarity analysis are presented and discussed. Experiments performed on university projects are presented. Experimental results include the distribution of similarity in C and C++ projects. Analysis of spectral similarity distribution identifies the most similar pairs of projects that can be considered as candidates for plagiarism.

Keywords. plagiarism detection, software comparison, clone detection, spectral analysis, code metrics

1 Introduction

The identification of similarity between code fragments can be based on comparing information derived from syntactic analysis or it can be based on character string comparison from the source code text. This paper presents plagiarism analysis performed by a tool called CLAN (CLone ANalyzer) that computes code fragments similarities by considering syntactically driven software metrics.

Similarity analysis using software metrics requires that each fragment be characterized by a set of features measured by metrics such as the number of passed parameters, the number of statements, cyclomatic complexity, and so on.

Metrics computation requires the parsing of source code to identify interesting fragments, which often are associated to software functions, and extract software metrics. For sake of completeness the details of each step are briefly summarized below.

Clones are defined as code fragments indistinguishable under a given criterion. Different granularities may be considered when extracting clone information (e.g., compound statement or function body). The process defined to study clone evolution is outlined in Figure 1 and consists of the following, subsequent phases:

Dagstuhl Seminar Proceedings 06301
Duplication, Redundancy, and Similarity in Software
http://drops.dagstuhl.de/opus/volltexte/2007/986

1. Handling of preprocessor directives;
2. Parsing and fragments identification;
3. Metrics extraction; and
4. Clone identification.

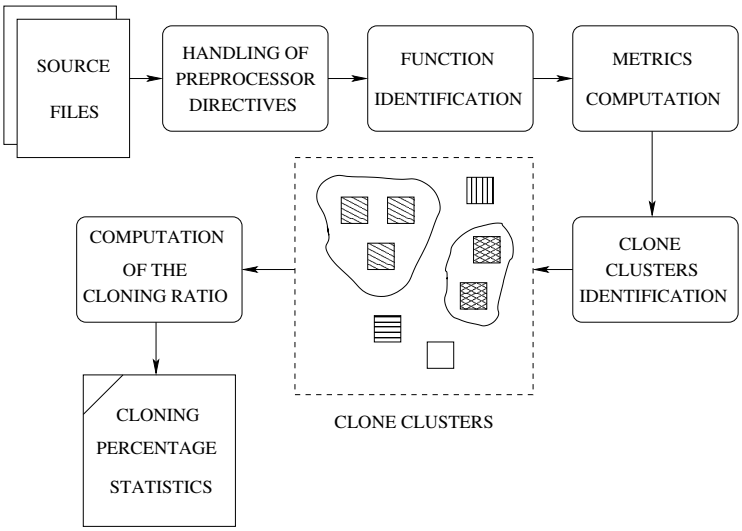

Fig. 1. The clone identification process.

Section 2 describes metrics based spectral similarity analysis. Section 3 discusses experiments and results, while conclusions can be found in Section 4.

2 Metrics Based Spectral Similarity Analysis

2.1 Parsing and fragments Identification

C and C++ systems are likely to encompass a variety of inter-mixed programming styles, programming patterns, idioms, coding standards and naming conventions. Most noticeably, both the ANSI-C and the Kernigham & Ritchie style will possibly be present. To localize and extract function definitions, we adopted an approach inspired by island-driven parsing: once islands (e.g., the function bodies or the signatures) were identified, the in-between code was scanned, and function definitions extracted by means of a hand-coded parser.

2.2 Metrics extraction

Following the approach proposed in [1], the functions extracted as illustrated above were compared on the basis of software metrics accounting for layout, size, control flow, function communication and coupling. In particular, each fragment was modeled by the following seven software metrics:

- number of function calls (CALLS)
- number of used or defined local variables (LOCALS)
- number used or defined non-local variables (NONLCALS)
- number of parameters (PARNUM)
- number of statements (STMNT)
- number of branches (NBRANCHES)
- number of loops (NLOOPS)

Metrics extraction can be performed in a time linear with respect to the number of fragments. Different sets of metrics could indeed be adopted (e.g., those used in [1]). However, we experienced that, on sufficiently large systems, the use of different sets of metrics does not significantly influence the results. Additionally, for the experiments on C++ programs, six metrics only have been used, since the number used or defined non-local variables was not used.

It is worth noticing that the proposed software metrics are not tied to a specific fragment similarity approach and those presented in [2,3,4,5,6,7,1,8] could be used in an almost interchangeable way.

Differently from the procedure customarily followed in the past, (e.g., in [1]), function names and file/unit names were not used as metrics.

2.3 Thresholds-based Quantized Matching Approach

Let:

$$fragments = (f_1, ..., f_M)$$

$$metrics = \begin{pmatrix} m_{1,1}, & ..., & m_{1,k} \\ m_{2,1}, & ..., & m_{2,k} \\ & ... & \\ m_{M,1}, & ..., & m_{M,k} \end{pmatrix} \tag{1}$$

$$thresholds = (th_1, ..., th_k)$$
$$clusters = \{cl_i \mid cl_i \in \mathcal{P}(fragments)\}$$

where $\mathcal{P}(fragments)$ is the power set of the set of fragments, be respectively the set of code fragments, the metrics matrix in which rows correspond to fragments and columns to metrics dimensions, the active dimensions on which similarity has to be measured, the matching thresholds for all dimensions, and the clusters of detected clones.

An original algorithm for threshold based clone quantization is presented in Figure 2, where:

$$\begin{array}{l} qClusters = \{cl_i \mid cl_i \in \mathcal{P}(fragments)\} \\ kSet : fragments \times \mathcal{N}^m,\ m \in \mathcal{N} \end{array} \tag{2}$$

Code fragments which belong to the same cluster returned by *computeQClusters* are considered similar under the given threshold sensitive criterion. Quantized clone clusters represent a partition of all fragments. In particular, fragments which belong to the same cluster identified by a key belonging to *kSet* satisfy the property that fragments corresponding to the metrics vector belong to the same hyper-parallelepiped identified by the following multi-dimensional interval:

$$\begin{array}{c} [kSet[f][1] \cdot th[1],\ (kSet[f][1]+1) \cdot th[1]] \times \\ \ldots \times \\ [kSet[f][i] \cdot th[i],\ (kSet[f][i]+1) \cdot th[i]] \times \\ \ldots \times \\ [kSet[f][n] \cdot th[n],\ (kSet[f][n]+1) \cdot th[n]] \end{array} \tag{3}$$

Execution time complexity of the clone quantization algorithm shown in Figure 2 is $\mathcal{O}(M \cdot n)$, where M is the number of fragments and n is the number of active dimensions. Since n is often kept constant with respect to the number of fragments M, once the matching strategy has been decided, and also often ($n \ll M$) and it is small, execution time complexity can be considered linear in terms of the number of fragments, i.e., it is $\mathcal{O}(M)$.

```
1   (qClusters, kSet) ← computeQClusters(fragments, metrics, thresholds)

2       qClusters.clear()
3       forall f ∈ fragments
4           i = 1
5           key = ()
6           while (i ≤ metrics[f].size())
6               if thresholds[i] > ε
7                   code = int(metrics[f][i] / thresholds[i]))
6               else
7                   code = int(metrics[f][i])
8               key.append(code)
9               i = i + 1
10          kSet[f] = key
11          qClusters[key] = qClusters[key] ∪ {f}
12      return (qClusters, kSet)
```

Fig. 2. Threshold-Based Clone Quantization

Figure 3 shows the original spectral analysis based on threshold clone quantization that is used to compute the similarity between two projects. The rationale for spectral analysis is that plagiarism is hard to deeply hide, if little programming energy is deployed. Surface differences are quickly ignored by thresholds of increasing levels

```
1   projectSimilarity ← computeSpectrum(fragments1, fragments2,
                                        metrics1, metrics2,
                                        thIncrements)

2       projectSimilarity = 0
3       for step ∈ [0..MAX_STEPS]
4           for dim ∈ [0..MAX_DIM]
5               curThresholds[dim] = step * thIncrements[dim]
6           (qClusters1, kSet1) = computeQClusters(fragments1, metrics1,
                                                   curThresholds)
7           (qClusters2, kSet2) = computeQClusters(fragments2, metrics2,
                                                   curThresholds)
8           common = 0
9           forall k ∈ kSet1 | k ∈ kSet2
10              common = common + | qClusters1[k] | + | qClusters2[k] |
10          projectSimilarity = projectSimilarity +
                                + common / ( | fragments1 | + | fragments2 | )
13      return projectSimilarity
```

Fig. 3. Spectral analysis

Quantizing the hyper-volume of the clone pseudo-space introduces a quantization error, which is reflected by the fact that two fragments in different clusters may indeed be closer than the threshold and therefore introduce false negatives in the clone analysis. However, in spectral similarity analysis, quantization error at step s and threshold $s \cdot th_i$ in dimension i is $(s \cdot th_i)$ / 2 and eventually cancels out at step $s' >= 2 \cdot s$ corresponding to threshold $s' \cdot th_i = 2 \cdot s \cdot th_i$.

A typical trend in spectral analysis is shown in Figure 4. The high quickly saturating curve is an example of the spectrum obtained by comparing two projects that show high similarity. At threshold 0 the two projects are not similar, but as the threshold increases, the similarity increases too and quite rapidly. Conversely, the lower curve represents two projects that are not similar. At null threshold similarity is null and it remains low for increasing threshold values. As expected, for high threshold values, also non-similar projects show high similarity values. What allows us to easily distinguish between the two cases, are not necessarily the absolute values of similarities, but rather the remarkably different trend of similarities with respect to increasing threshold values. Spectral analysis computes an approximation of the surface underlying a given similarity curve.

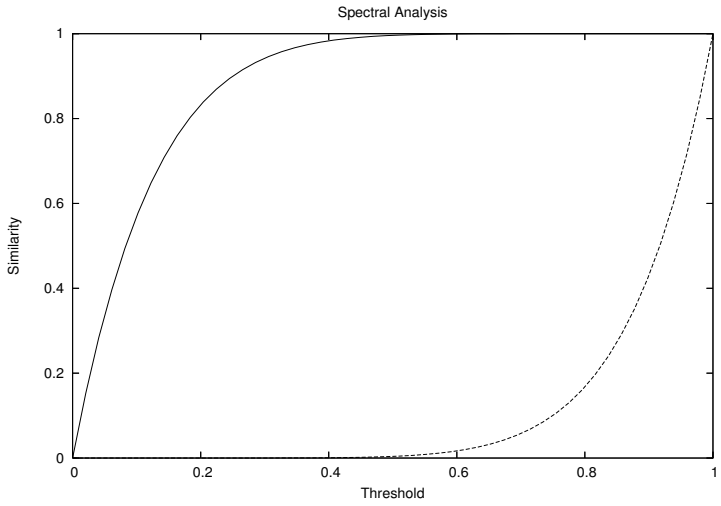

Fig. 4. Typical spectra

Figure 5 shows the algorithm to compute spectral similarity on all pairs of projects. The algorithm computes the spectral similarity between all pairs of different projects and returns the sorted by similarity list of all pairs of projects.

```
1  similarityList ← computeProjectSimilarity(projects, thIncrements)

2      forall p_i ∈ projects
3          forall p_j ∈ projects | j > i
4              similarity = computeSpectrum(fragments_i, fragments_j,
                                            metrics_i, metrics_j,
                                            thIncrements)
5              similarityList.add(similarity, i, j)
6      similarityList.sort()
7      return similarityList
```

Fig. 5. Plagiarism analysis

Complexity of algorithm in Figure 5 is quadratic on the number of distinct projects. The quadratic complexity can be reduced to a linear one by using the algorithm in Figure 6 that implements an approximated approach based on astrophysics modeling of galaxies and that has been described in [9] in details. The algorithm computes the center of mass of projects based on their fragments'

metrics. Spectral analysis is subsequently performed by clustering centers of mass descriptions along different incremental threshold levels.

```
1  similarityList ← computeGalacticProjectSimilarity(projects,
                                                      thIncrements)
2      forall p_i ∈ projects
3          projectSimilarity[p_i] = 0
4          descr[i] ← centerOfMass(p_i)
5      for step ∈ [0..MAX_STEPS]
6          for dim ∈ [0..MAX_DIM]
7              curThresholds[dim] = step * thIncrements[dim]
8          (qClusters, kSet) = computeQClusters(projects, descr[i],
                                                curThresholds)
9          forall k ∈ kSet
10             forall p_k ∈ qClusters[k]
11                 projectSimilarity[p_k] = projectSimilarity[p_k] +
                                            + | qClusters[k] |
12     forall p_j ∈ projects
13         similarityList.add(projectSimilarity[p_j], p_j)
14     similarityList.sort()
15     return similarityList
```

Fig. 6. Galactic Plagiarism Analysis

Similarity list as computed by the *galactic* spectral similarity analysis algorithm reported in Figure 6 returns the pairs of similar projects sorted by similarity order.

Complexity of *galactic* spectral similarity analysis is linear on the number of projects and the linear factor is affected by the number of metric dimensions and by the number of steps. Often the number of metric dimensions and the number of steps are much smaller than the number of projects, so the algorithm can be thought as linear on the number of projects.

3 Experiments and Discussion

Figure 7 shows the results obtained after similarity analysis of pairs of projects written in C language, while Figure 7 shows the results obtained for projects written in C++. Results have been computed using the algorithm in Figure 5.

Metrics	Threshold increments
CALLS	1
LOCALS	1
NONLCALS	1
PARNUM	1
STMNT	3
NBRANCHES	1
NLOOPS	1

Table 1. Metrics and threshold increments

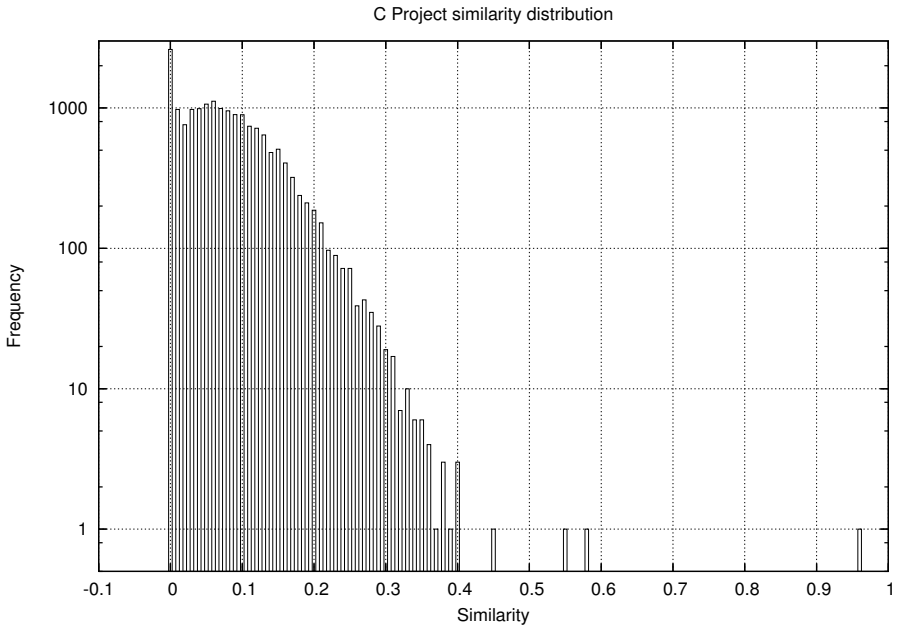

Fig. 7. C project distribution

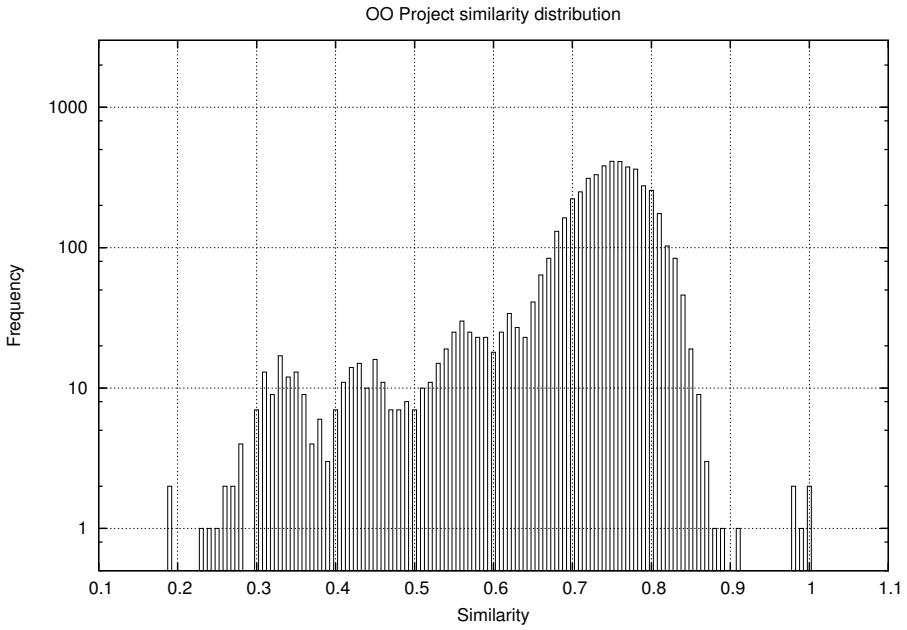

Fig. 8. C++ project distribution

Similarity contrast is very good for procedural code, while distribution of similarity for C++ code is less sharp. This can be partially explained since reference classes were given to all students as a part of the projects, so C++ projects are biased towards higher average similarity than C projects. Furthermore, C++ methods tend to be smaller than average natural C functions and so chances of similarity in terms of similar metrics value are more often occurring.

To sharpen the contrast between C++ and C class structure should be investigated and taken into consideration and also inter-class relationship could be studied.

Outliers of similarity distribution in the high end were identified so that the most similar projects under metrics-based spectral analysis were identified.

A DP-matching based project pairs alignment and visualization process [7] was performed on most similar projects to ease the human inspection of similarities.

No hypothesis about the causes of similarity were made, but identified cases were transmitted to the university administration for follow up.

4 Conclusions

A metrics based plagiarism detection approach in an academic environment has been presented The presented approach has been successfully used to discourage plagiarism in course projects

CLAN is a very fast, small memory, accurate, but conservative tool. The authors are quite happy with the proven CLAN performance in terms of speed and also of robustness to parse and analyze code.

CLAN can be recommended in applications were speed and precision make its use advantageous. We can think of routinely performed analyses during development and of interactive environments possibly to support agile development. Also, when very large applications need to be processed CLAN family of tools which includes the "galactic clone detection" [9] for object oriented systems is appropriate for scalability.

Acknowledgment

The research on CLAN has been funded in part by the National Sciences and Engineering Research Council of Canada and Bell Canada.

References

1. Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press, Monterey, CA (1996) 244–253
2. Baker., B.: On finding duplication and near-duplication in large software systems. In: Proceedings of the Working Conference on Reverse Engineering. (1995)
3. Baxter, I., Yahin, A., Moura, l., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press. (1998) 368–377
4. Buss, E., De Mori, R., Gentleman, W., Henshaw, J., Johnson, H., Kontogiannis, K., Merlo, E., Muller, H., Mylopoulos, J., Paul, S., Prakash, A., Stanley, M., Tilley, S., Troster, J., Wong, K.: Investigating reverse engineeering technologies for the cas program understanding project. IBM Systems Journal **33** (1994) 477–500
5. Johnson, J.H.: Identifying redundancy in source code using fingerprints. In: CASCON. (1993) 171–183
6. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28** (2002) 654–670
7. Kontogiannis, K., De Mori, R., Bernstein, R., Galler, M., Merlo, E.: Pattern matching for clone and concept detection. Journal of Automated Software Engineering **3** (1996) 77–108
8. McCabe, T.: Reverse engineering, reusability, redundancy: the connection. American Programmer **3** (1990) 8–13
9. Merlo, E., Antoniol, G., DiPenta, M., Rollo, F.: Linear complexity object-oriented similarity for clone detection and software evolution analysis. In: Proceedings of the International Conference on Software Maintenance - IEEE Computer Society Press, IEEE Computer Society Press (2004) 412–416